Lecture 9 - Testing

What is testing?
V-Model of testing
What to test
Types of tests
Testing strategies
Testing automation
Testing roles in companies
Fault tracking
Tracking (roundup, testtrack pro etc)

What is testing?

- Testing is a process to help identify
  - correctness, completeness, security, quality, reliability, etc.
  - in fact to determine if the software is useful for its intended purpose
- Determine if things work as expected
  - Documentation, Processes, Programs, etc.
- "Program testing can be used to show the presence of bugs, but never to show their absence!" E.D.Dijkstra

Test Process Maturity

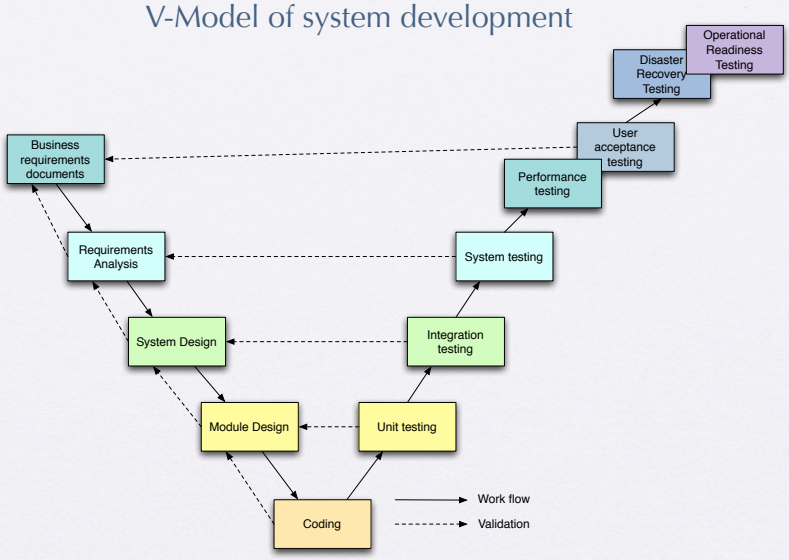
- **Level 0** - There's no difference between testing and debugging
- **Level 1** - The purpose of testing is to show that the software works
- **Level 2** - The purpose of testing is to show that software doesn't work
- **Level 3** - The purpose of testing is not to show anything specific, but to reduce the risk of using the software
- **Level 4** - Testing is a mental discipline that helps all IT professionals develop higher quality software

- Ref. Boris Beizer, Software Testing Techniques, 2nd edition 1990.

Validation and verification

- Common terms in testing
- Validation
  - Testing to see if the system meets user requirements
  - "Did we build the right product?"
- Verification
  - Testing to see if the construction *process* is done properly
  - "Did we build the product correctly?"

V-Model of system development



Faults, Errors,Defects, Failures and Debugging

- We test to detect failures - a difference in a system's observed behaviour from its expected behaviour
- When a failure is discovered you have to classify its urgency
  - Colours not displayed correctly
  - AI in a computer game not as good as hoped
  - ATM dispensing incorrect amounts of cash
- An error is an incorrect internal state cased by a fault.
- Failures are caused by defects
  - in the software - typically a design mistake
  - in the requirements
- If a failure is found you can do Defect Locating - identifing where the failure origin

Ref. Introduction to Software Testing, p Ammann, J Offutt

Faults, Defects, Debugging

- Defect locating
  - Requires analytical skills - often the most scientific skill in "computer science".
  - Postulate a cause, propose and perform an experiment, check the result.
- Once located you still have to decide what to do. You may...
  - decide to leave it unchanged
  - fix it in the next release
  - fix it immediately
- Debugging covers
  - Defect location
  - Fixing the problem
  - Is less nuanced than the discussion above
  - Is separate from testing

Testing - Attributes of tests

- Traceability
  - Every test should be linkable back to a requirement.
- Completeness
  - Each requirement (functional or non functional) should have at least one test associated with it (and generally more).
- Coverage
  - Every possible path through the system should have a test

Testing

- Use cases specify the behaviour of a system
- This means you should come up with tests for
  - the main flow of each use case
  - the alternative flows in a use case
- These are used during *User acceptance testing*.

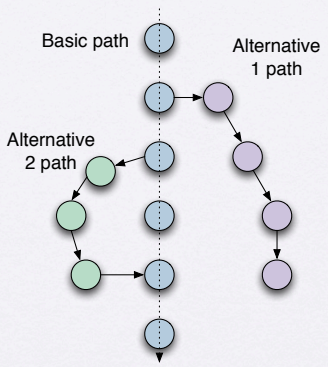


Diagram from http://www.ibm.com/developerworks/rational/library/edge/08/feb08/sheldon_lenters/

Testing - what can it tell us?

- Testing can:-
  - Show errors in coding
  - Give us greater confidence in our software (but software can still be discontinuous)
  - Measure performance
  - Indicate quality
- Testing is NOT a substitute for quality, but it can reveal poor quality systems

What to test?

- You should test everything (if you are concerned about it being correct!)
  - Documentation
    - translations!
  - Procedures
    - Installation
    - Fault logging & tracking
    - Payment
  - Programs
    - Unit testing
    - Integration testing
    - System testing
  - Backups!
    - Do you back up your computer?
    - Do your backups work?

What to test?

- Every aspect in the development of a computer system can be tested
- Requirements
  - Do they represent reality/can they be physically implemented?
  - Have previous systems with these requirements been built/did they have problems (feasibility analysis)
  - Are they self-consistent?
  - Check with the user
- Analysis
  - Are the relationships really present in the problem?
  - Is it possible to meet all requirements with the model?

What to test?

- Design
  - Will the system be able to implement the system within given time constraints?
  - Can the design be built within budget?
- Implementation
  - Does each class work as specified?
  - Do the classes work together properly?
- Documentation
  - Can the user understand it, and use it to solve problems?
  - Does it match the functionality of the system?

Testing artefacts

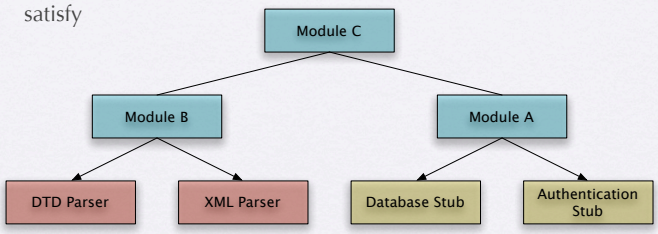
- Testing results in the following being created...
  - Test case
    - Description of a single test
      - id
      - link back to design/requirements documents
      - preconditions
      - algorithm
      - input/ouput expected
  - Test script
    - A test case, test procedure and test data
    - Can be manual or automatically created/run
  - Test suite
    - collection of tests

Unit Testing

- Unit testing
  - Checking low level modules work in isolation
  - Modules typically work with other modules, so we have to create a framework to support the modules.
  - Modules that are called can be replaced with simple versions that are easy to code up, and return values that allow you to test sufficiently.
  - Similarly you may need to write code to call the module you are testing. This is often called a test harness.

Integration testing

- Once each unit has been tested, we start combining units to form larger groups.
- Integration testing tests whether the components communicate with each other properly, and provide appropriate behaviour.
- Need to establish pre/post conditions/invariants
- Need to identify requirements that the component is meant to satisfy



System testing

- System testing tests all of the components together as they are expected to be delivered to the customer.
- These tests are based against the use cases and scenarios created described during requirements.

Testing strategies

- The best tests are those that discover faults.
  - As we can't test for every possibility we have to choose some tests over others, and hope they are representative.
  - We have to have a measure by which we can choose tests
  - This requires an understanding of what faults are likely.
  - Different fault classes can be tested in different ways.
- The most common types software errors
  - not knowing the language
  - typographical errors
  - errors in problem solving logic
  - algorithms that provide poor results in some cases
  - single or critical input values that yield unanticipated results
  - data structure defects in design or translation of the design
  - misinterpretation of specifications

Not knowing the language

- In 1990, a bug in the New York telephone network switching software caused a cascading failure.
- The programmer didn't understand "break" properly.

In the switching software (written in C), there was a long
"do . . . while" construct, which contained
a "switch" statement, which contained
an "if" clause, which contained a
"break," which was intended for
the "if" clause, but instead broke from
the "switch" statement.

Typographical errors

- Errors where mistyping or accidentally choosing the wrong symbol.
- C is very vulnerable as many character sequences are so similar to other valid sequences.

```
if (a = b) - should be if (a==b) ?

 printf ("a &  b is %d\n", a & b);
printf ("a && b is %d\n", a && b);

   & and && are very different!
```


Algorithms that provide poor results in some cases

- Computing with floating point numbers is especially vulnerable to the order in which items are manipulated.
- See program "Algorithm.java".

Single or critical input values that yield unanticipated results

- USS Yorktown was experimenting with a computer controlled system for propulsion.
- A Microsoft database stored information for the control systems
- A user entered in 0 into a field that wasn't validated, resulting in a divide by 0 error.
- The failure spread, and locked the entire system.
- The ship had to be towed back to port.

Testing for common errors

- We need to check that common problems are addressed in our tests.
- Some tests are done by the compiler - e.g. passing correct parameters to a function, checking assignment statements, checking syntax.
- Some languages are better than others at providing support for this.
  - if (a = b) {…}
  - C doesn't prevent this, but Java (mostly) does.
- Primitive languages like C often require the help of static code checking tools, such as lint.

Testing for common errors

- Some tests can be made at run time.
- Java provides for array bounds detection, but not overflow detection for integers.
- Java cannot provide range checks (e.g. if a "month" variable goes outside the range 1..12)
- Java checks for null pointers
- Primitive languages such as C require the addition of runtime checking tools such as Purify which checks...
  - memory leaks
  - sources of Segmentation Faults
  - places where array bounds are exceeded
  - places where the program tries to follow Null pointers

Two types of testing

- We can examine the modules (classes) in two ways
- White box testing
  - Write tests for the internals of the class - the if statements, while statements, algorithms etc.
  - Perform code reviews/write the code in pairs. Involves a second person checking your program before you start performing run-time tests.
- Black Box testing
  - Ignores the internal implementation details and concentrates on the service that the modules offer. For example tests pre/post conditions and invariants. Also tests sequences of calls into the object. For example for a stack object check that sequences of push/pop results in correct behaviour.

White box testing

- White box testing looks in detail at the code that makes up a function. Typical places to inspect are branching points where decisions are made...
  - if statements, loops, multiway branches (switch statements).
- Boundary conditions for variables can also be tested. For example, a variable to hold a month value should be checked for being outside of range.
- Boundary conditions for types, such as integer overflow conditions, should also be checked.

Testing for common faults

- “Bugs lurk in corners and congregate at boundaries”
- Boundary conditions - those areas where the behaviour of the system changes dramatically from one state to another - are those that have the greatest potential for error.
- The behaviour of the if statement below changes when “a” has values of 9 and 10. A test case should ensure that both “a” is set to both values.
- Here we don’t test the if, but the actions. Sometimes we test the logic of the if (especially when it is more complicated).

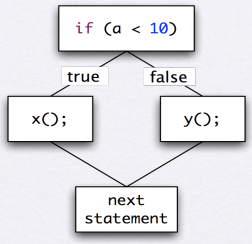
```
if (a < 10) {
     x ();
  } else {
     y ();
     }
```

```
Should be tested with ‘a’ set to
            9, 10, 11
```

Coverage testing

- Every line of code in a program should be tested at least once.
- This requires that all paths through a program are followed.
- We can build a graph of the code we are testing to highlight this.
- How do you test exceptional situations?

```
if (a < 10) {
     x ();
  } else {
     y ();
     }
```



Coverage testing

- Coverage testing can be complicated when you have to deal with an object's history.
  - An object retains the values of its attributes between method calls, so testing involves both
    - the order method calls are made
    - well as the value of parameters.
- Depending on which constructor was called, count will be either 0 or a user supplied value.

```
public class Test {
   in count = 0;

   public Test (int count) {
      this.count = count;
      this.list = new Vector ();
   }

   public Test () {
      this.list = new Vector ();
   }
```


Automating testing

- Software needs to be tested every time any change is made.
- Small changes can have unforeseen effects, so it is wise to check all of the software.
- The modern method of software development encourages lots of small changes to ensure software is always kept runnable.
- This results in lots of testing - the only solution is to automate the process.
- Automated testing can help identify problems earlier.

How useful is testing?

- "The most powerful weapon in your toolbox is comprehensive unit testing"
- It's this kind of thinking that causes too many of the problems you are trying to prevent.
- According to “Programming Productivity” by Capers Jones,
  - “Unit testing will only find 25% of the defects. At its best, unit testing finds 50% of the problems. Compare this to, for example, formal inspections, which find 60% of defects on average, 70% at best. Modelling and prototyping (build one to throw away) can find up to 80% of defects.”
- None of these methods will build a completely defect-free program, but by combining a number of them, you can get extremely close (99%)

(author unknown)

Testing roles

- Some companies separate the role of testing from that of developer because
  - people are blind to the problems in their own code
  - integration testing requires testing the interaction between developer’s code
  - the job of testing is too large for one person
  - The developer still does preliminary testing to ensure their code works to spec.
  - The code is then released to the testers who run their own testing on it.
- The human aspects of a separate testing team often results in trouble
  - Testers are often the last in line in the product release cycle. Any delay from the developers means they work rush hours under extreme pressure

Testing roles

- The following are likely on a testing project
  - Tester
  - Test Designer
  - Test/QA Team Lead
  - Test Build Manager
  - Test Configuration Manager
  - Test automater/automation developer
  - Test administrator

Fault tracking

- If you want to correct faults you need to
  - identify them (result of testing, or general use)
  - record what their status is
  - identify what versions of software they occur in
- Fault tracking software help automate this
  - Roundup
  - TestTrackPro

How much testing should you do?

- Eventually testing has to come to an end. How do you decide when that is?
- You need to establish release criteria...
  - You need to define criteria that you can check before agreeing to release
  - Define success for your software
  - Software may contain faults, but it should provide value for your customers
  - Determine what your company needs
  - Ensure the criteria are, like requirements, "SMART" - Specific, Measurable, Attainable, Relevant, Trackable

Example release criteria

- All code must compile and build for all platforms
- Zero high priority bugs
- For all open bugs, documentation in release notes with workarounds
- All planned QA tests run, at least ninety percent pass
- Number of open defects decreasing for last three weeks.
- Feature x unit tested by developers, system tested by QA, verified with customers A, B before release
- Ready to release by June 1

Ref http://www.jrothman.com/Papers/releasecriteria.html

Summary

- Testing is never a substitute for developing software with a good process.
- Testing is used to identify a program's divergence from the requirements
- Not all faults need to be fixed.
- Automated testing simplifies the process of testing, and can help development.